VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ24

Tým xgeierd00, varianta TRP-izp

Vedoucí týmu:	Daniel Geier – xgeierd00 – 25%
Další členové:	Patrik Mokruša – xmokrup00 – 25%
	Ariana Tomen – xtomen00 – 25%
	Klára Johanesová – 253003 – 25%

Obsah

1 Úvod **2**

2 Práce v týmu **2**
2.1 Rozdělení práce 2

3 Implementace překladače **2**
3.1 Lexikální analýza 2
3.2 Syntaktická analýza 3
3.2.1 Struktura stromu 3
3.2.2 Struktura uzlů derivačního stromu 3
3.3 Sémantická analýza 4
3.4 Tabulka symbolů 4
3.5 Generování kódu 5

4 Závěr **5**

5 Přílohy **6**
5.1 Diagram konečného automatu lexikální analýzy 6
5.2 LL-gramatika 7
5.3 Precedenční tabulka 8
5.4 LL-tabulka 9

6 Reference **10**

1 Úvod

Cílem projektu bylo vytvořit v týmu překladač imperativního jazyka IFJ24, který je podmnožinou jazyka Zig. Celý projekt je implementován v jazyce C. Dokumentace zahrnuje 4 přílohy, které obsahují Diagram konečného automatu lexikální analýzy, LL-gramatiku, Precedenční tabulku a LL-tabulku.

2 Práce v týmu

Projekt jsme vypracovali jako čtyřčlenný tým. Pro správu verzí jsme využili platformu Github. Komunikace probíhala na platformě Discord kde jsme měli časté setkání.

2.1 Rozdělení práce

- **Daniel Geier** – Generátor kódu, testování částí překladače
- **Patrik Mokruša** – Syntaktický analyzátor (rekurzivní sestup a precedenční syntaktická analýza), sémantický analyzátor, implicitní konverze ve výrazech, generátor kódu (struktura rekurzivní funkce), pomocné funkce v tabulce symbolů
- **Ariana Tomen** – Tabulka symbolů, zpracování výrazů sémantické analýzy, knihovna string
- **Klára Johanesová** – Lexikální analyzátor, dokumentace

3 Implementace překladače

Implementace překladače je rozdělena na 4 části. To jsou lexikální analýza, syntaktická analýza a sémantická analýza, a generování kódu. Každá z těchto částí je nezbytná pro správnou implementaci překladače.

Celková struktura projektu je rozdělena na 3 kroky. První je tvorba derivačního stromu, syntaktická analýza a naplnění globální tabulky symbolů. Druhý je průchod derivačním stromem při sémantické analýze. Třetí je druhý průchod derivačním stromem při generování kódu. Každá tato fáze je popsána v následujících kapitolách.

3.1 Lexikální analýza

Jako první část jsme implementovali lexikální analyzátor. Je implementován v souboru `scanner.c`, který je implementován na základě konečného stavového automatu.

Lexikální analýza funguje na principu, který načítá vstupní znaky a převede je na jednotlivé tokeny podle automatu. Tokeny jsou deklarovány v hlavičkovém souboru `scanner.h`. Jsou rozděleny do kategorií jako operátory, logické operátory, závorky, interpunkční znaky, klíčová slova, hodnoty, chyby a EOF. Každý token má vlastní strukturu a obsahuje typ a kategorii.

Hlavní funkce lexikálního analyzátoru je `scan`, je to velký `switch`, ve kterém vytváří tokeny podle jeho stavu. Bílé znaky a komentáře jsou ignorovány. Lexikální analyzátor následně pošle vytvořené tokeny syntaktickému analyzátoru pro další zpracování. Pokud funkce načte znak, který ne-

souhlasí s jazykem, tak je vrácena chybová hláška 1 a skončí. Podrobnosti o konečném stavovém automatu, jsou přiloženy v sekci přílohy.

3.2 Syntaktická analýza

Syntaktická analýza se skládá ze dvou hlavních částí, rekurzivního sestupu a precedenční analýzy.

Při analýze rekurzivním sestupem se rovnou vytváří derivační strom, který reprezentuje odvozování pravidel z LL-gramatiky. Každý uzel odpovídá terminálnímu nebo neterminálnímu uzlu a počet jeho potomků je určen pravou stranou pravidla produkce. Kořen stromu odpovídá výchozímu neterminálu (`<start>`) a listy reprezentují terminální uzly nebo epsilonové produkce.

3.2.1 Struktura stromu

- Každý uzel má 0 až 4 potomky (např. `node->first, node->second, ...`)
- Pokud pravidlo vede na epsilon, uzel nemá žádné potomky
- Strom se postupně rozšiřuje podle pravidel gramatiky, přičemž uzly se tvoří dynamicky

3.2.2 Struktura uzlů derivačního stromu

Každý uzel stromu je reprezentována strukturou `Node`:

- Typ uzlu (`NodeType`): Identifikuje, jaký neterminální nebo terminální uzel reprezentuje (např. `VariableDefine_N, FuncBody_N`)
- Ukazatel na potomky (`Node *first, Node *second, ...`)
  - Uzel může mít až 4 potomky podle pravidel gramatiky
  - Například uzel `FuncDefine_N` má čtyři potomky odpovídající pravidlu:
    `<id>, <params_define>, <data_type>, <func_body>`
- Data (`Data_value data`): Pro specifická data uzlů (identifikátory, datové typy, hodnoty, realace, ...)

Precedenční analýza je implementovaná v souboru `expressionparse.c`.

Ve funkcích `Parse_expression` a `reduce`, při redukcích tvoří derivační strom od spodu nahoru.

U semantické analýzy byl problém s rozšířením FUNEXP (expression v parametru volanné funkce). Zavedl jsem jenom jeden stack pro všechny rekurzivní volání, kde jednotlivé analýzy jsou oddělenné akorát terminály $.

Ošetření chyb je důležitá část syntaktikého analyzátoru. Při alokaci každého nového uzlu derivačního stromu je jeho ukazatel uložen do zásobníku, který je uložen jako globální proměnná. Pokud je během analýzy nalezena syntaktická chyba, smaže se celý zásobník s uzly a následně pošle chybovou hlášku. Musel jsem zavést tento zásobník, protože vytvářím derivační strom pomocí return hodnot funkcí v RS.

Narazili jsme na hlavní problém, který byl konflikt v LL-tabulce. Vyřešili jsme ho zavedením dočasného ukládání tokenů, pomocí `TokenBuffer` což umožňuje jednoduchý přístup k tokenům dopředu. Používá funkci `buffercheckfirst` pro konzumování a kontrolu typu tokenu. Některé konflikty se řeší podíváním se na token dopředu a některé se řeší voláním funkce v rekurzivním sestupu, která je schopná řešit více pravidel ll-gramatiky.

3.3 Sémantická analýza

Sémantická analýza se zaměřuje na ověřování správnosti významu programu, konkrétně na kontrolu typů, kontrolu definice proměnných a platnosti výrazů. Je implementovaná v souboru `semantic.c`.

Hlavní funkcí pro provádění sémantické analýzy je `semantic_scan`, která prochází rekurzivně derivační strom, a pokud narazí na uzel, jehož typ má ve `switch case`, pak zkontroluje jeho vlastnosti podle zadání.

Pro správu identifikátorů a jejich atributů používá sémantická analýza globální symbolickou tabulku a lokální tabulku symbolů, které si rekurzivně předává v parametru funkce `semantic_scan` a postupně do ní doplňuje informace.

Pro kontrolu výrazů, které se objevují ve výrazech a operacích, používá sémantická analýza funkci `semantic_expr`, která je deklarovaná v souboru `semantic.c`. Funkce také konvertuje číselné literály jak bylo specifikované v zadání projektu. Konverze literálů probíha od spodu nahoru a to tak, že rekurzivně projde derivační podstrom expression, narazí na operátor a zkontroluje, jestli se shodují. Pokud ano, nic se neděje, pokud ne, konvertuje celý podstrom na požadovanný typ.

Při ošetření chybových stavů se v globální proměnné uchovává kořen derivačního stromu. Při nalezení chyby se zavolá funkce `free_parse_tree`, která uvolní alokovanou paměť. Po uvolnění se ukončí proces s odpovídajícím chybovým kódem.

3.4 Tabulka symbolů

Pro implementaci tabulky symbolů jsme s týmem zvolili variantu TRP. Tento přístup zahrnuje použití hashovací tabulky s otevřenou adresací.

Tabulka je implementována jako dynamické pole, které používá dvojí hashovací funkcí. Tento přístup nám umožňuje efektivně spravovat symboly, i když tabulka roste, zaručuje efektivní hledání, vkládání a mazání symbolů. První hashovací funkce provádí počáteční výpočet hash hodnoty a druhá pomáhá při řešení kolizí. Tento přístup garantuje, že tabulka bude efektivní i při větších množstvích dat.

Pro zajišťování správné funkci tohoto přístupu je nutné, aby velikost tabulky byla prvočíslem, což je realizované pomocí funkcí `is_prime` a `get_next_prime` (pokud nebyla počáteční hodnota prvočíslem). To výrazně zlepšuje rovnoměrné rozložení položek a snižuje počet kolizí.

Každá položka v tabulce symbolů je reprezentována strukturou **SymbolTableEntry**, která uchovává klíč (identifikátor), typ symbolu (proměnná nebo funkce) a data o proměnné nebo funkci. Tato struktura umožňuje uchovávat různé informace, jako jsou datové typy, parametry funkcí a stav symbolů (obsazen nebo smazán).

Během návrhu tabulky symbolů docházelo často ke změnám struktury kvůli rozdílům v představách o tom, jak by měla fungovat a jakou funkci by měla v našem projektu plnit. Nakonec se díky komunikaci v týmu podařilo tento problém vyřešit.

Speciální funkce implementace:

Pro každou funkci používáme spojený seznam parametrů. To nám umožňuje flexibilně přidávat parametry k funkcím bez nutnosti přerozdělování paměti při každém přidání.

Tabulka symbolů používá metodu dynamického zvětšení, kdy se její velikost zvětší dvakrát, pokud je zaplněná více než na 75%.

Pro mazání symbolů používáme logické mazání. Symboly nejsou fyzicky odstraněny okamžitě, ale označeny jako smazané, aby se předešlo poškození struktury dat při opětovném použití polí.

3.5 Generování kódu

Celý kód je implementován v souboru `generator.c` a ve hlavičkovém souboru `generator.h`. Jedná se o průběžnou generaci bez použití AST (Abstraktní syntaktický strom), kde vycházíme z vytvořeného derivačního stromu. Rekurzivně se prochází a průběžně generujeme jednotlivé konstrukce a logiky. Jednotlivé hodnoty následně bereme v daném uzlu podle LL gramatiky.

Je implmentován jako jedna velká funkce `switch case`, kde ke každému typu uzlu je odpovídající `case`, kde se vygeneruje na standardní výstup. Následně obsahuje zvlášť funkce pro generovaní výrazů, které probíhají na zasobníku. Funkce pro převod řetězce na escape sekvenci pro ifj mezikód a funkce na jednotné vygenerovaní builtin funkcí jazyka. Výsledky funkcí se předávají přes globální proměnnou. Parametry funkce se vypisují pozpátku.

V generátoru předpokládáme že je kód validní a tudíž neřešíme některé podmínky a chybové hlášky, které řeší ostatní části překladače. Generujeme rovnou podle vygenerovaného stromu z LL gramatiky. Každý uzel podle LL gramatiky odkazuje na patřičné potomky, ze kterých bere data.

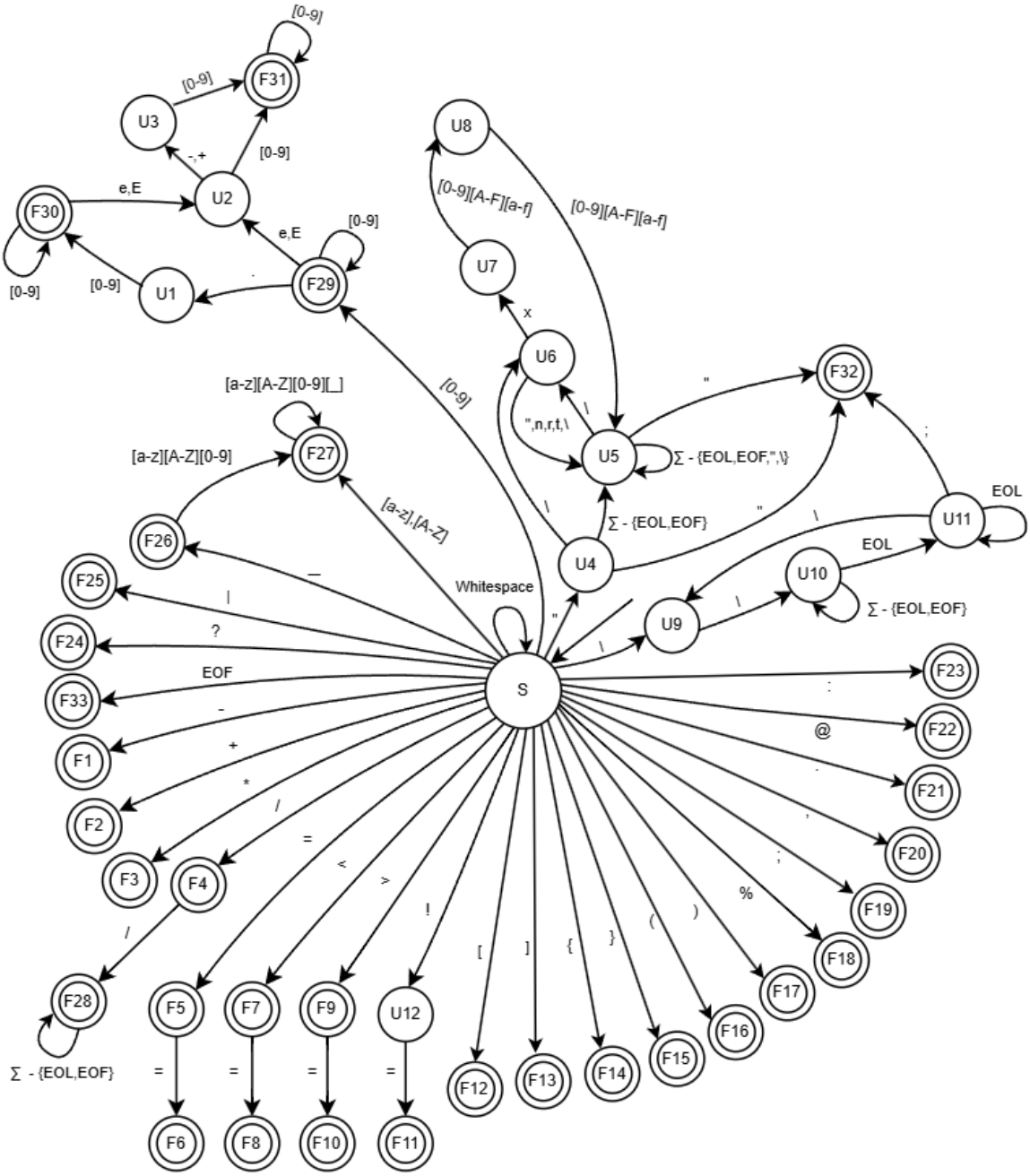
Problémy při implementaci nastaly v zamezení redefinice ve funkci `while` a řesení funexp. Následně byly vyřešeny a opraveny.

4 Závěr

Tento projekt se nám podařil dokončit a umožnil nám si vytvořit vlastní funkční překladač. Vyzkoušeli jsme si vytvořit projekt od návrhu až po testování. Práce na projektu nám dala mnoho zkušeností a nových vědomostí. Naučili jsme se používat různé nástroje, jako je Git pro správu verzí, a nástroje pro analýzu a testování kódu. Zároveň jsme si vylepšili schopnosti ve vzájemné spolupráci v týmu. Shodli jsme se že nám tento projekt dal velmi mnoho, nejen z hlediska technických dovedností, ale i ve smyslu týmové práce a schopnosti řešit složité problémy.

5 Přílohy

5.1 Diagram konečného automatu lexikální analýzy



F1: T_Minus	F9: T_Greater	F18: T_Percent	F27: T_ID\KW
F2: T_Plus	F10: T_Great_Eq	F19: T_SemiC	F28: T_Comment
F3: T_Mul	F11: T_Not_Eq	F20: T_Comma	F29: T_Integer
F4: T_Div	F12: T_L_Square_B	F21: T_Dot	F30: T_Float
F5: T_Assign	F13 T_R_Square_B	F22: T_At	F31: T_Exponent
F6: T_Equal	F14: T_L_Curly_B	F23: T_Colon	F32: T_String
F7: T_Lesser	F15: T_R_Curly_B	F24: T_Question	F33: T_EOF
F8: T_Less_Eq	F16: T_L_Round_B	F25: T_Pipe	
	F17: T_R_Round_B	F26: T_Underscore	S: Starting state

5.2 LL-gramatika

	PRAVIDLA	PREDICT
1)	<start> → <program_prolog><program>	const
2)	<program_prolog> → const ifj = @import("ifj24.zig");	const
3)	<program> → <func_define><program>	pub
4)	<program> → ε	EOF
5)	<id> → id	id_token
6)	<int> → int	int_token
7)	<float> → float	float_token
8)	<string> → string	str_token
9)	<data_type> → i32	i32
10)	<data_type> → f64	f64
11)	<data_type> → []u8	[
12)	<data_type> → void	void
13)	<func_define>→pub fn <id>(<params_define>) <data_type>{<func_body>}	pub
14)	<params_define> → ε	)
15)	<params_define> → <id>: <data_type><params_define_next>	id
16)	<params_define_next> → ε	)
17)	<params_define_next> → ,<id>: <data_type><params_define_next>	,
18)	<func_body> → ε	}
19)	<func_body> → <statement><func_body>	const, var, id, if, while, return, _
20)	<statement> → <variable_define>	const, var
21)	<statement> → <variable_assign>	id, _
22)	<statement> → <if>	if
23)	<statement> → <while>	while
24)	<statement> → <void_call>	id
25)	<statement> → <return_statement>	return
26)	"params_rhs"→ <EXPRESSION>	int, float, id, (
27)	"params_rhs"→ <string>	str
28)	"rhs"→ <EXPRESSION>	int, float, id, (
29)	"rhs"→ <string>	str
30)	<variable_define> → const <id> : <data_type> = "rhs";	const
31)	<variable_define> → var <id>: <data_type> = "rhs";	var
32)	<variable_assign> → <id> = "rhs";	id
33)	<variable_assign> → _ = "rhs";	_
34)	<func_call> → <id>(<params>)	id
35)	<params> → ε	)
36)	<params> → "params_rhs"<params_next>	id, str, float, int, (
37)	<params_next> → ε	)
38)	<params_next> → , "params_rhs"<params_next>	,
39)	<if> → if (<EXPRESSION>){<func_body>}else{<func_body>}	if
40)	<if> → if (<id>) |<id>| {<func_body>}else{<func_body>}	if
41)	<while> → while (<EXPRESSION>){<func_body>}	while
42)	<while> → while (<id>) |<id>| {<func_body>}	while
43)	<void_call> → <id>(<params>);	id
44)	<return_statement> → return;	return
45)	<return_statement> → return "rhs";	return
46)	<EXPRESSION> → "zpracovano pomoci precedencni analyzy"	

5.3 Precedenční tabulka

	EXPR.	comp.	term	factor	>	>=	<	<=	==	!=	+	-	*	/	float	int	id	(	)	$
EXPRESSION																				
compared																				
term																				
factor																				
>					>	>	>	>	>	>	<	<	<	<	<	<	<	<	>	>
>=					>	>	>	>	>	>	<	<	<	<	<	<	<	<	>	>
<					>	>	>	>	>	>	<	<	<	<	<	<	<	<	>	>
<=					>	>	>	>	>	>	<	<	<	<	<	<	<	<	>	>
==					>	>	>	>	>	>	<	<	<	<	<	<	<	<	>	>
!=					>	>	>	>	>	>	<	<	<	<	<	<	<	<	>	>
+					>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
-					>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
*					>	>	>	>	>	>	>	>	>	>	<	<	<	<	>	>
/					>	>	>	>	>	>	>	>	>	>	<	<	<	<	>	>
float					>	>	>	>	>	>	>	>	>	>					>	>
int					>	>	>	>	>	>	>	>	>	>					>	>
id					>	>	>	>	>	>	>	>	>	>					>	>
(					<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	
)					>	>	>	>	>	>	>	>	>	>					>	>
$					<	<	<	<	<	<	<	<	<	<	<	<	<	<		accept

5.4 LL-tabulka

Tokeny →	O	[	]	{	}	(	)	%	;	,	.	:	?	|	_	const	else	fn	if	i32	f64	null	pub	return	u8	var	void	while	import	id	Int	float	EOF	string
start																1																		
program_prolog																2																	
program																			3													4	
id																													5				
int																														6			
float																															7		
string																																	8
data_type		11													9	10										12							
func_define																						13											
params_define							14																						15				
params_define_next							16			17																							
func_body			18												19	19			19					19		19			19				
statement															21	20			22					25		20		23		21,24				
params_rhs						26																							26	26	26		27
rhs						28																							28	28	28		29
variable_define																30										31							
variable_assign															33														32				
func_call																													34				
params						36	35																						36	36	36		36
params_next							37			38																							
if																			39,40														
while																											41,42						
void_call																													43				
return_statement																							44,45										

Pozn. O - Operátory

6 Reference

1. Meduna, A., Lukáš, R.: *IFJ, přednášky [online]*. Brno, Dostupné z:
   `http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/index.php`
2. Wikipedia: *Hašovací tabulka*. Dostupné z:
   `https://en.wikipedia.org/wiki/Hash_table`
3. Burgetová, I., Hranický, R., Honzík, J.M.: *IAL, přednášky [online]*. Brno.